XML

HO GENT

This document will learn you how to parse and process XML in SQL Server. You will learn how to treat XML as a data type in SQL Server. You can parse any of the SQL Server string data types, such as [n][var]char, [n]text, varbinary,and image, into the xml data type by casting (CAST) or converting (CONVERT) the string to the xml data type. Untyped XML is checked to confirm that it is well formed. If there is a schema associated with the xml type, validation is also performed.

XML documents can be encoded with different encodings (for example, UTF-8, UTF-16, windows-1252). The following outlines the rules on how the string and binary source types interact with the XML document encoding and how the parser behaves.

Since nvarchar assumes a two-byte unicode encoding such as UTF-16 or UCS-2, the XML parser will treat the string value as a two-byte Unicode encoded XML document or fragment. This means that the XML document needs to be encoded in a two-byte Unicode encoding as well to be compatible with the source data type. A UTF-16 encoded XML document can have a UTF-16 byte order mark (BOM), but it does not need to, since the context of the source type makes it clear that it can only be a two-byte Unicode encoded document.

The content of a varchar string is treated as a one-byte encoded XML document/fragment by the XML parser. Since the varchar source string has a code page associated, the parser will use that code page for the encoding if no explicit encoding is specified in the XML itself If an XML instance has a BOM or an encoding declaration, the BOM or declaration needs to be consistent with the code page, otherwise the parser will report an error.

The content of varbinary is treated as a codepoint stream that is passed directly to the XML parser. Thus, the XML document or fragment needs to provide the BOM or other encoding information inline. The parser will only look at the stream to determine the encoding. This means that UTF-16 encoded XML

needs to provide the UTF-16 BOM and an instance without BOM and without a declaration encoding will be interpreted as UTF-8.

If the encoding of the XML document is not known in advance and the data is passed as string or binary data instead of XML data before casting to XML, it is recommended to treat the data as varbinary. For example, when reading data from an XML file using OpenRowset(), one should specify the data to be read as a varbinary(max) value:

```
select CAST(x as XML)
from OpenRowset(BULK 'filename.xml', SINGLE_BLOB) R(x)
```

SQL Server internally represents XML in an efficient binary representation that uses UTF-16 encoding. User-provided encoding is not preserved, but is considered during the parse process.

The following example casts a string variable that contains an XML fragment to the xml data type and then stores it in the xml type column:

```
CREATE TABLE T(c1 int primary key, c2 xml)
go
DECLARE  @s varchar(100)
SET @s = '<Cust><Fname>Andrew</Fname><Lname>Fuller</Lname></Cust>'
```

The following insert operation implicitly converts from a string to the xml type:

```
INSERT INTO T VALUES (3, @s)
```

You can explicitly cast() the string to the xml type:

```
INSERT INTO T VALUES (3, cast (@s as xml))
```

Or you can use convert(), as shown in the following:

```
INSERT INTO T VALUES (3, convert (xml, @s))
```

In the following example, a string is converted to xml type and assigned to a variable of the xml data type:

```
declare @x xml
declare  @s varchar(100)
SET @s = '<Cust><Fname>Andrew</Fname><Lname>Fuller</Lname></Cust>'
set @x =convert (xml, @s)
select @x
```

In general:
XML instances can be stored in a variable or a column of xml type.

Syntax:

```
xml ( [ CONTENT | DOCUMENT ] xml_schema_collection )
```

CONTENT
Restricts the xml instance to be a well-formed XML fragment. The XML data can contain multiple zero or more elements at the top level. Text nodes are also allowed at the top level. This is the default behavior.

DOCUMENT
Restricts the xml instance to be a well-formed XML document. The XML data must have one and only one root element. Text nodes are not allowed at the top level.

xml_schema_collection
Is the name of an XML schema collection. To create a typed xml column or variable, you can optionally specify the XML schema collection name.

Using the SELECT Statement with a FOR XML Clause
You can use the FOR XML clause in a SELECT statement to return results as an XML instance. You can store this in a variable (or column) of type xml. For example:

```
DECLARE @xmlDoc xml
SET @xmlDoc = (SELECT Column1, Column2
                FROM   Table1, Table2
                WHERE   Some condition
                FOR XML AUTO)
 ...
```

The SELECT statement returns a textual XML fragment that is then parsed during the assignment to the xml data type variable.

You can also use the TYPE directive in the FOR XML clause that directly returns a FOR XML query result as xml type:

```
Declare @xmlDoc xml
SET @xmlDoc = (SELECT ProductModelID, Name
               FROM   Production.ProductModel
               WHERE  ProductModelID=19
               FOR XML AUTO, TYPE)
SELECT @xmlDoc
```

This is the result:

```
<Production.ProductModel ProductModelID="19" Name="Mountain-100" />...
```

In the following example, the typed xml result of a FOR XML query is inserted into an xml type column:

```
CREATE TABLE T1 (c1 int, c2 xml)
go
INSERT T1(c1, c2)
SELECT 1, (SELECT ProductModelID, Name
           FROM Production.ProductModel
           WHERE ProductModelID=19
           FOR XML AUTO, TYPE)
SELECT * FROM T1
go
```

The following example inserts a constant string into an xml type column:

```
CREATE TABLE T(c1 int primary key, c2 xml)
INSERT INTO T VALUES (3,
'<Cust><Fname>Andrew</Fname><Lname>Fuller</Lname></Cust>')
```

XML Declaration
The XML declaration in an instance is not preserved when the instance is stored in the database. For example:

```
CREATE TABLE T1 (Col1 int primary key, Col2 xml)
GO
```

```
INSERT INTO T1 values (1, '<?xml version="1.0" encoding="windows-1252"
?><doc></doc>')
GO
SELECT Col2
FROM T1
```

The result is <doc/>.

Load

You can bulk load XML data into the server by using the bulk loading capabilities of SQL Server, such as bcp. OPENROWSET allows you to load data into an XML column from files. The following example illustrates this point.

This example shows how to insert a row in table T. The value of the XML column is loaded from file C:\MyFile\xmlfile.xml as CLOB, and the integer column is supplied the value 10.

```
INSERT INTO T
SELECT 10, xCol
FROM    (SELECT *
    FROM OPENROWSET (BULK 'C:\MyFile\xmlfile.xml', SINGLE_CLOB)
 AS xCol) AS R(xCol)
```

For XML (in SQL Server)

A SELECT query returns results as a rowset. You can optionally retrieve formal results of a SQL query as XML by specifying the FOR XML clause in the query. The FOR XML clause can be used in top-level queries and in sub queries. The top-level FOR XML clause can be used only in the SELECT statement. In sub queries, FOR XML can be used in the INSERT, UPDATE, and DELETE statements. It can also be used in assignment statements.

In a FOR XML clause, you specify one of these modes:

- **RAW**
- **AUTO**
- **EXPLICIT**
- **PATH**

The RAW mode generates a single <row> element per row in the rowset that is returned by the SELECT statement. You can generate XML hierarchy by writing nested FOR XML queries.

The AUTO mode generates nesting in the resulting XML by using heuristics based on the way the SELECT statement is specified. You have minimal control over the shape of the XML generated. The nested FOR XML queries can be written to generate XML hierarchy beyond the XML shape that is generated by AUTO mode heuristics.

The EXPLICIT mode allows more control over the shape of the XML. You can mix attributes and elements at will in deciding the shape of the XML. It requires a specific format for the resulting rowset that is generated because of query execution. This rowset format is then mapped into XML shape. The power of EXPLICIT mode is to mix attributes and elements at will, create wrappers and nested complex properties, create space-separated values (for example, OrderID attribute may have a list of order ID values), and mixed contents.

The PATH mode together with the nested FOR XML query capability provides the flexibility of the EXPLICIT mode in a simpler manner.

These modes are in effect only for the execution of the query for which they are set. They do not affect the results of any subsequent queries.

FOR XML is not valid for any selection that is used with a FOR BROWSE clause.

Example:

```
USE AdventureWorks2014
GO
SELECT Cust.CustomerID,
       OrderHeader.CustomerID,
       OrderHeader.SalesOrderID,
       OrderHeader.Status
FROM Sales.Customer Cust
```

```
INNER JOIN Sales.SalesOrderHeader OrderHeader
ON Cust.CustomerID = OrderHeader.CustomerID
FOR XML AUTO
```

In the example, the SELECT statement retrieves information from the Sales.Customer and Sales.SalesOrderHeader tables in the AdventureWorks2014 database. This query specifies the AUTO mode in the FOR XML clause.

Computed columns & functions

If queries are made principally on a small number of element and attribute values, you may want to promote those quantities into relational columns. This is helpful when queries are issued on a small part of the XML data while the whole XML instance is retrieved. Creating an XML index on the XML column is not required. Instead, the promoted column can be indexed. Queries must be written to use the promoted column. That is, the query optimizer does not target again the queries on the XML column to the promoted column.

The promoted column can be a computed column in the same table or it can be a separate, user-maintained column in a table. This is sufficient when singleton values are promoted from each XML instance. However, for multi-valued properties, you have to create a separate table for the property, as described in the following section.

A computed column can be created by using a user-defined function that invokes xml data type methods. The type of the computed column can be any SQL type, including XML. This is illustrated in the following example.

Create the user-defined function for a book ISBN number:

```
CREATE FUNCTION udf_get_book_ISBN (@xData xml)
RETURNS varchar(20)
BEGIN
   DECLARE @ISBN   varchar(20)
   SELECT @ISBN = @xData.value('/book[1]/@ISBN', 'varchar(20)')
   RETURN @ISBN
```

```
END
```

Add a computed column to the table for the ISBN:

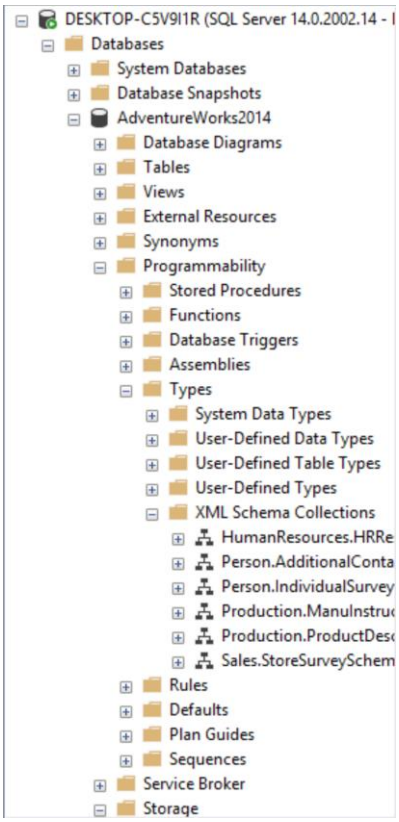
```
ALTER TABLE      T
ADD   ISBN AS dbo.udf_get_book_ISBN(xCol)
```

```
Declare @xmlDoc xml
SET @xmlDoc = (SELECT ProductName, Price
               FROM   Product
               WHERE  ProductTypeID=1
               FOR XML AUTO, TYPE)
SELECT @xmlDoc
```

XML Schemas
XML Schemas can be added to the catalogue of a database. To add a Schema to a DB, execute a DDL statement:

```
CREATE XML SCHEMA COLLECTION [Person].[IndividualSurveySchemaCollection] AS
N'<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:t="http://schemas.microsoft.com/sqlserver/2004/07/adventure-
works/IndividualSurvey"
targetNamespace="http://schemas.microsoft.com/sqlserver/2004/07/adventure-
works/IndividualSurvey" elementFormDefault="qualified
...
</xsd:schema>'
```

Schemas can afterwards be found at: *Programmability > Types > XML Schema Collections*



An XML Schema describes the structure of an XML document. The XML Schema language is also referred to as XML Schema Definition (XSD).The purpose of an XML Schema is to define the legal building blocks of an XML document:

1. the elements and attributes that can appear in a document
2. the number of (and order of) child elements
3. data types for elements and attributes
4. default and fixed values for elements and attributes

When sending data from a sender to a receiver, it is essential that both parts have the same "expectations" about the content.

With XML Schemas, the sender can describe the data in a way that the receiver will understand.

A date like: "03-11-2004" will, in some countries, be interpreted as 3.November and in other countries as 11.March.
However, an XML element with a data type like this:

<date type="date">2004-03-11</date>
ensures a mutual understanding of the content, because the XML data type "date" requires the format "YYYY-MM-DD".

Example of a Schema:

```
<?xml version="1.0"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">

<xs:element name="note">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="to" type="xs:string"/>
      <xs:element name="from" type="xs:string"/>
      <xs:element name="heading" type="xs:string"/>
      <xs:element name="body" type="xs:string"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>

</xs:schema>
```

An XML schema provides the following:

- Validation constraints. Whenever a typed xml instance is assigned to or modified, SQL Server validates the instance.
- Data type information. Schemas provide information about the types of attributes and elements in the xmldata type instance. The type information provides more precise operational semantics to the values contained in the instance than is possible with untyped xml. For example, decimal arithmetic operations can be performed on a decimal value, but not on a string value. Because of this, typed XML storage can be made significantly more compact than untyped XML.

In T-SQL and to process XML, Schemas can be used to create a typed XML data type. XML instances can be stored in a variable or a column of xml type.

Syntax:

```
xml ( [ CONTENT | DOCUMENT ] xml_schema_collection )
```

CONTENT
Restricts the xml instance to be a well-formed XML fragment. The XML data can contain multiple zero or more elements at the top level. Text nodes are also allowed at the top level. This is the default behavior.

DOCUMENT
Restricts the xml instance to be a well-formed XML document. The XML data must have one and only one root element. Text nodes are not allowed at the top level.

xml_schema_collection
Is the name of an XML schema collection. To create a typed xml column or variable, you can optionally specify the XML schema collection name.

Example of a typed XML data type: The Schema/type is `Person.IndividualSurveySchemaCollection`

```
USE AdventureWorks;
GO
DECLARE @DemographicData xml (Person.IndividualSurveySchemaCollection);
SET @DemographicData =  (SELECT TOP 1 Demographics FROM Person.Person);
SELECT @DemographicData;
```

A result of the above select can be this below. This result is typed against a Schema.

```
<IndividualSurvey
xmlns="http://schemas.microsoft.com/sqlserver/2004/07/adventure-
works/IndividualSurvey">
  <TotalPurchaseYTD>-25</TotalPurchaseYTD>
  <DateFirstPurchase>2003-12-01Z</DateFirstPurchase>
  <BirthDate>1940-07-25Z</BirthDate>
  <MaritalStatus>S</MaritalStatus>
  <YearlyIncome>50001-75000</YearlyIncome>
  <Gender>M</Gender>
  <TotalChildren>3</TotalChildren>
  <NumberChildrenAtHome>0</NumberChildrenAtHome>
  <Education>Bachelors </Education>
  <Occupation>Management</Occupation>
  <HomeOwnerFlag>1</HomeOwnerFlag>
  <NumberCarsOwned>2</NumberCarsOwned>
  <CommuteDistance>10+ Miles</CommuteDistance>
</IndividualSurvey>
```

The Schema is:

```
CREATE XML SCHEMA COLLECTION [Person].[IndividualSurveySchemaCollection]
AS N'<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:t="http://schemas.microsoft.com/sqlserver/2004/07/adventure-
works/IndividualSurvey"
targetNamespace="http://schemas.microsoft.com/sqlserver/2004/07/adventure-
works/IndividualSurvey" elementFormDefault="qualified">
 <xsd:element name="IndividualSurvey">
   <xsd:complexType>
     <xsd:complexContent>
       <xsd:restriction base="xsd:anyType">
         <xsd:sequence>
           <xsd:element name="TotalPurchaseYTD" type="xsd:decimal"
minOccurs="0" />
           <xsd:element name="DateFirstPurchase" type="xsd:date" minOccurs="0"
/>
           <xsd:element name="BirthDate" type="xsd:date" minOccurs="0" />
           <xsd:element name="MaritalStatus" type="xsd:string" minOccurs="0"
/>
           <xsd:element name="YearlyIncome" type="t:SalaryType" minOccurs="0"
/>
           <xsd:element name="Gender" type="xsd:string" minOccurs="0" />
           <xsd:element name="TotalChildren" type="xsd:int" minOccurs="0" />
           <xsd:element name="NumberChildrenAtHome" type="xsd:int"
minOccurs="0" />
           <xsd:element name="Education" type="xsd:string" minOccurs="0" />
           <xsd:element name="Occupation" type="xsd:string" minOccurs="0" />
           <xsd:element name="HomeOwnerFlag" type="xsd:string" minOccurs="0"
/>
           <xsd:element name="NumberCarsOwned" type="xsd:int" minOccurs="0" />
           <xsd:element name="Hobby" type="xsd:string" minOccurs="0"
maxOccurs="unbounded" />
           <xsd:element name="CommuteDistance" type="t:MileRangeType"
minOccurs="0" />
           <xsd:element name="Comments" type="xsd:string" minOccurs="0" />
         </xsd:sequence>
       </xsd:restriction>
     </xsd:complexContent>
   </xsd:complexType>
 </xsd:element>
 <xsd:simpleType name="MileRangeType">
   <xsd:restriction base="xsd:string">
     <xsd:enumeration value="0-1 Miles" />
     <xsd:enumeration value="1-2 Miles" />
     <xsd:enumeration value="2-5 Miles" />
     <xsd:enumeration value="5-10 Miles" />
     <xsd:enumeration value="10+ Miles" />
   </xsd:restriction>
 </xsd:simpleType>
 <xsd:simpleType name="SalaryType">
   <xsd:restriction base="xsd:string">
     <xsd:enumeration value="0-25000" />
     <xsd:enumeration value="25001-50000" />
     <xsd:enumeration value="50001-75000" />
```

```
      <xsd:enumeration value="75001-100000" />
      <xsd:enumeration value="greater than 100000" />
    </xsd:restriction>
  </xsd:simpleType>
</xsd:schema>'
```

1. Exercises

1. Complete the following steps

Download books.xml from Chamilo and execute this code. This piece of code will create a new table and will load existing XML in the table. From then, you can process the XML in T-SQL.

```
CREATE TABLE T (col1 INT PRIMARY KEY, xCol XML);
GO
INSERT INTO T
SELECT 10, xCol
FROM    (SELECT *
    FROM OPENROWSET (BULK 'C:\My-Path\Databanken II\books.xml', SINGLE_CLOB)
 AS xCol) AS R(xCol)
GO
SELECT xCol
FROM T
```

For example XQUERY can then be used to process your XML document. To execute an XQUERY command in T-SQL do:

`SELECT xmlColumn.query('xquery cmd')`

This XQUERY command returns all authors as an XML document

```
SELECT xCol.query('/catalog/book/author')
FROM T
```

This XQUERY command returns all unique authors as individual values.

```
SELECT xCol.query('distinct-values(/catalog/book/author)')
FROM T
```

This XQUERY command returns all books with a price lower than 30, as an XML document. price is an XML element.

```
SELECT xCol.query('/catalog/book[price<30]')
FROM T
```

This XQUERY command returns the book with book id bk102, as an XML document. id is an XML attribute.

```
SELECT xCol.query('/catalog/book[@id="bk102"]')
FROM T
```

XQUERY is outside the scope of this course, but can be used to process and query XML. In T-SQL, you would need a column or variable of type xml for this.

2. Create a temporary table used for processing XML in T-SQL. There will be one column: field1 from type XML. All further exercises are on DB XTREME.

3. Fill this table (column) with XML: transform the content of a table (Product) to XML. Select only the name and price of all products with productTypeId = 1. Add this XML to the table (field1).

4. Use XQUERY to query field1 of your temp table. Return (XML document) all books with a price higher than 1000.